NASA Technical Paper 1200

User's Guide for SFTRAN/1100

William F. Ford and Theodore E. Fessler

APRIL 1978

NASA

NASA Technical Paper 1200

User's Guide for SFTRAN/1100

William F. Ford and Theodore E. Fessler
Lewis Research Center
Cleveland, Ohio

National Aeronautics
and Space Administration

**Scientific and Technical
Information Office**

1978

USER'S GUIDE FOR SFTRAN/1100

by William F. Ford and Theodore E. Fessler

Lewis Research Center

SUMMARY

Extensions and improvements have been made to SFTRAN, a structured-programming language. This improved language has been implemented as a precompiler that translates from SFTRAN to FORTRAN. It has been available to batch and conversational users of the Lewis Research Center's UNIVAC 1100 computer system for the past year. This report describes the SFTRAN language and its use.

Conversational Time-Sharing System (CTS) command subroutines have been implemented that eliminate the complications of dealing with extra files and processing steps that the use of a precompiler would otherwise require. These command subroutines are described and their use is illustrated by examples.

INTRODUCTION

In recent years, two new programming concepts have received a good deal of attention in the literature. The first of these may be loosely termed "GO-TO-less programming," although a more appropriate description might be "avoidance of numbered statements." When this concept is employed, program flow is controlled by constructs, or structures, that imply a certain flow-chart function; the name given each construct is suggestive of its function. A principal benefit of GO-TO-less programming is that the resulting code is easier to read, both because the structure names are more meaningful than numbers and also because the reader's eye is not forced to leap around on the page or from one page to another.

The second concept involves what has been referred to as "stepwise refinement," the process of successively refining one's description of the solution method in terms of ever-more-primitive components or processes. At each level of refinement, only enough detail is presented to make the method clear. Anything of a complicated nature is merely referred to, with its precise definition postponed until later. Sometimes called top-down programming, this technique also results in an easier-to-read code.

Only a few ideas need be kept in mind at each level of description; and the big, important ones can be put first, at the top, where they belong.

The combination of these two principles, GO-TO-less programming and stepwise refinement, results in what we will denote as "structured programming." The combined technique has more to offer, however, than merely good readability. It also provides a natural sequence of steps to be taken in the programming process: It replaces much of the art of programming with a methodology that is easy for the novice to learn and very efficient in the hands of the experienced. Also, a high degree of program modularity is obtained, which means that the code produced can easily be changed, extended, or adapted for other uses.

This report concerns a new implementation of the programming language SFTRAN, which was created by John T. Flynn[1] for the specific purpose of providing a language suitable for structured programming. Flynn's implementation of his SFTRAN language was a precompiler that translated from SFTRAN to FORTRAN. This permitted a great degree of simplification in the precompiler program: It only had to recognize the few special structures that control program flow. Also, by translating to FORTRAN, the benefits of program portability were automatically obtained.

Our work retains all of Flynn's original structures. The new SFTRAN precompiler differs from Flynn's in that it has been given additional language features, has had some operating limitations removed, and has been designed to run more efficiently. This report describes the SFTRAN language and its use, as newly implemented.

Our effort, however, goes beyond refinement of an existing precompiler. We extend the concepts of modularity and top-down development to the area of task management and provide a set of command subroutines for this purpose. Thus, the programmer is able to select one of several jobs, and one of several parts of that job, and to invoke one of several operations to be performed on that part. This can all be done by means of simple, brief statements designed expressly for the purpose. The programmer is thus freed of concern for bothersome details of an operational nature and can instead concentrate his attention on those areas where his skill and effort are of maximum benefit. These task-management subroutines, as implemented on the Lewis Research Center's UNIVAC 1100 computer system, are described in this report. Examples of their use are included.

SFTRAN LANGUAGE

In brief, SFTRAN (Structured ForTRAN) is a programming language with the following features:

[1]SFTRAN User Guide, JPL Interoffice Computing Memorandum 337 (Section 914), July 31, 1973.

(1) It eliminates the burden of dealing with statement numbers. Program sections are referred to by name, not number.

(2) It allows and even encourages the grouping of instructions into small, natural units within a program or subprogram. These units can be given unique, descriptive names and are displayed in listings in a manner that makes their internal structure immediately apparent to the eye.

(3) It looks very much like ordinary FORTRAN, except for the manner in which branching and looping are handled.

The SFTRAN language is implemented by a precompiler that generates FORTRAN source code from SFTRAN source code. SFTRAN programs can therefore be considered machine independent to the same degree that their FORTRAN translations are machine independent.

The basic structures by means of which SFTRAN avoids the GO-TO or implied GO-TO statements of FORTRAN are the following:

DO-PROCEDURE

IF-THEN

IF-THEN-ELSE

DO-CASE

DO-FOR

DO-UNTIL

DO-WHILE

DO-WITH

The first structure refers to the PROCEDURE, a group of statements of any kind to which a unique name is given. (This name may and should be descriptive, of arbitrary length, with embedded blanks, special symbols, etc., if desired, and is the only means by which the group of statements may be invoked. The PROCEDURE thus operates very much like a baby subroutine within the program body, except that it can refer to any of the variables of the program.) The remaining seven structures provide commonly required types of program control, in which some sort of decision-making is performed.

One feature of the SFTRAN precompiler is the option to automatically assign statement numbers to normal SFTRAN output, flagging the various SFTRAN statements. Also, statement numbers that were supplied by the programmer in order to flag normal FORTRAN statements can be stripped off by the precompiler or left intact, as desired. (These statement numbers make it possible to debug directly from the SFTRAN code. The FORTRAN code produced by the precompiler is unsuitable for this, because it is

hard to read.) Later on, when the programs are in good shape and clean text is desired, the SFTRAN precompiler can be invoked again, with these options reversed, to produce the final listings.

Programming in SFTRAN

Any group of statements and structures, providing it has only one entrance and one exit, can be designated as a single structural entity. This is accomplished in SFTRAN by means of the PROCEDURE declaration, which assigns a unique, descriptive name to the entity. At any point or points within the program, this entity can be invoked and executed by using the DO-PROCEDURE statement. The process of creating arbitrarily complex code, therefore, becomes one of organizing an appropriate assembly of concepts whose functions are indicated by their names but whose precise definition is deferred until the necessary level of detail is reached.

In those regions of a program where the flow of commands is not purely sequential, some sort of transfer is required. The transfer may be lateral, as when one of several alternatives must be selected and performed; or it may be backward, as when a block of instructions must be repeated a number of times. These cases are termed branching and looping, respectively.

The simplest form of branching involves only one block of instructions and the decision whether or not to perform it. In SFTRAN this is accomplished by means of the structure IF-THEN. When there are two alternatives, the modification IF-THEN-ELSE may be used. For more than two alternatives, a different form of structure, the DO-CASE, is available.

The simplest form of looping involves repeating a block of instructions a certain number of times while an index is incremented. In SFTRAN this is accomplished by means of the structure DO-FOR. In more general cases the block must be repeated until a certain condition is attained or while a certain condition holds; these forms of looping are implemented by the DO-UNTIL and the DO-WHILE structures, respectively. Still more general cases are treated with the DO-WITH structure, in which the condition test (either UNTIL or WHILE) may be placed anywhere within the block to be repeated.

Once the branching and looping structures have been defined, any program, no matter how complex, can be reduced to a set of simple structures each of which has only one entrance and one exit. As such, the structures are comparable to the simple statements that form sequential code.

Basic Structures

Each SFTRAN structure is delimited by a keyword statement that marks its beginning and an END statement that marks its completion; other keyword statements may occur between these two. The keyword statements and the END statement form the skeleton of the structure.

The remaining statements comprise the body of the structure and may be chosen freely by the programmer. For clarity, the body of a structure is indented in the output listing (but not the source) relative to the skeleton of that structure. (This principle continues to hold even when one structure forms part of the body of another structure.)

DO-PROCEDURE. - Any set of sequential statements and structures, providing it has only one entrance and one exit, may be designated as a single structural entity called a PROCEDURE. Its beginning is indicated by a keyword PROCEDURE statement, and its finish by an END statement. The keyword statement also contains the name assigned to the PROCEDURE - a string of characters contained within parentheses. An example of such a structure is

```
PROCEDURE (VECTOR PRODUCT: A(I) = B(I) X C(I) )
   A(1,I)=B(2,I)*C(3,I)-B(3,I)*C(2,I)
   A(2,I)=B(3,I)*C(1,I)-B(1,I)*C(3,I)
   A(3,I)=B(1,I)*C(2,I)-B(2,I)*C(1,I)
END
```

In this example the PROCEDURE name is chosen to be as descriptive as possible. The name may be any length; all characters, including blanks, are significant. But PROCEDURE names should not contain apostrophes or unmatched (left with right) parentheses.

Loosely speaking, a PROCEDURE may be regarded as a subprogram that is internal to the program or subprogram in which it appears. It is called from another point in the same program by a DO-PROCEDURE statement; for the preceding example,

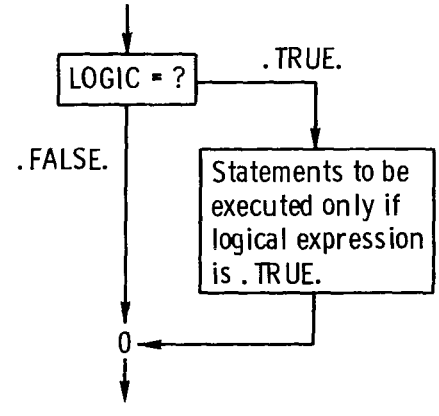
```
      .
      .
      .
DO (VECTOR PRODUCT: A(I) = B(I) X C(I) )
      .
      .
      .
```

There are a few simple rules governing the use of PROCEDURES. To begin with, the keyword PROCEDURE statement must stand alone; it cannot be defined within another structure. It must have no means of entry except for calls by DO-PROCEDURE state-

ments; these DO statements are only effective within the program or subprogram in which the PROCEDURE is defined.

The name in a PROCEDURE call must be exactly the same as the name in the PROCEDURE definition. For this reason, a PROCEDURE name may appear in only one keyword PROCEDURE statement; it may appear more than once in DO-PROCEDURE calls, but it must appear at least once. Finally, although the body of a PROCEDURE may be made up of other SFTRAN statements and structures, including calls to other PROCEDURES, it may not call itself either directly or indirectly; PROCEDURES do not have recursive capability. The SFTRAN precompiler does not check this; it is the programmer's responsibility to ensure that recursive calls will not occur.

IF-THEN. - The simplest form of branching is the IF-THEN structure, whose flow diagram is



An example of IF-THEN coding is

```
        .
        .
        .
IF (.NOT.FOUND) THEN
   DO (REPORT MISSING ITEM)
   STOP
END
        .
        .
        .
```

(Of the two statements forming the body of the IF-THEN structure, the first is an SFTRAN-type statement and the second is a FORTRAN-type statement.) An abbreviation of this structure is possible whenever only one statement is contained in the conditional block. For example, instead of

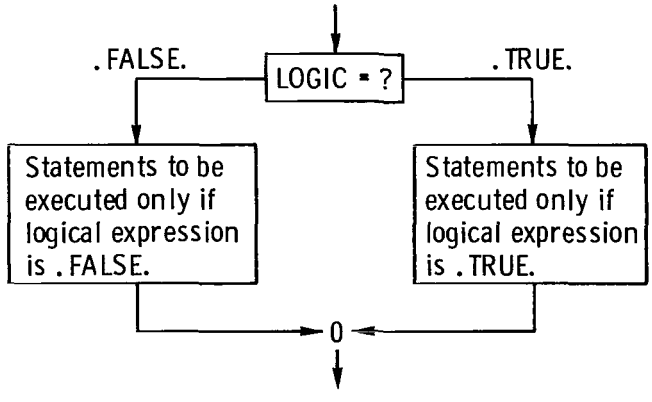
```
        .
        .
        .
IF (.NOT.FOUND) THEN
   DO (REPORT MISSING ITEM)
END
        .
        .
        .
```

the abbreviation

```
        .
        .
        .
IF (.NOT.FOUND) DO (REPORT MISSING ITEM)
        .
        .
        .
```

may be used. Note that THEN and END are omitted in the abbreviated version, an exception to the general rule that every structure begins with a keyword and ends with an END statement.

IF-THEN-ELSE. - When branching involves two alternatives, the structure IF-THEN-ELSE may be used. Its flow diagram is

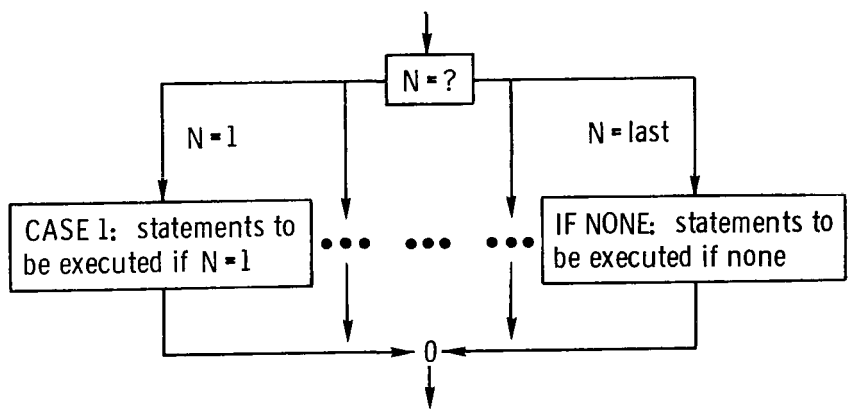


An example of IF-THEN-ELSE coding is

```
      .
      .
      .
IF (I.EQ.J) THEN
   DO (I=J CASE)
   CALL SUB1(X,Y,Z)
ELSE
   DO (NORMAL CASE)
   CALL SUB2(X,Y,Z)
END
      .
      .
      .
```

The statements to be executed if LOGIC is .TRUE. come right after the IF-THEN keyword statement; the statements to be executed if LOGIC is .FALSE. come right after the ELSE keyword statement. In either case, program flow then passes to the first statement following the structure's END statement.

DO-CASE. - When branching involves more than two alternatives, the DO-CASE structure may be used. Here the alternative to be chosen is determined by examining an integer variable rather than a logical variable. The flow diagram of the DO-CASE structure is



An example of DO-CASE coding is

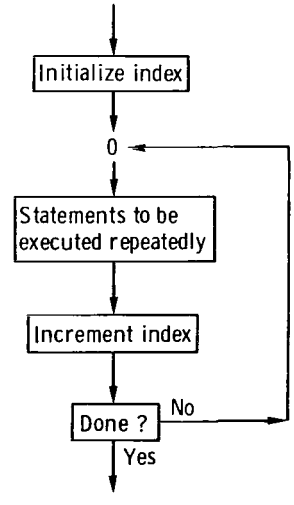
```
      .
      .
      .
DO CASE (ITYPE,3)
CASE 1
   POLY=A*X+B
CASE 2
   POLY=A*X**2+B*X+C
CASE 3
   DO (NOT LINEAR OR QUADRATIC)
IF NONE
   DO (REPORT TYPE ERROR)
END
      .
      .
      .
```

The case-choice and the case-limit appear in the keyword DO CASE statement. The definitions of each possible case form the body of the structure, separated by keyword CASE N statements. As usual, an END statement marks the completion of the structure.

The case-choice must be a nonsubscripted integer variable; and the case-limit must be a positive, literal, integer constant. The number of keyword CASE N statements must be equal to the case-limit, and they must be given in sequential order, beginning with CASE 1. Each case definition follows its own keyword CASE N statement; even if a definition is null, containing no statements, the keyword CASE N is required.

If the case-choice is not within range of the number of cases defined (less than 1 or greater than the case-limit), control will pass to the next statement after the END of the DO-CASE structure unless an IF NONE contingency case is provided. If an IF NONE case is provided, it must follow the last case in the structure.

DO-FOR. - The simplest form of looping involves repeating a block of instructions a certain number of times, while an index is incremented. This is accomplished by means of the DO-FOR structure, whose flow diagram is



An example of DO-FOR coding is

```
        .
        .
        .
DO FOR I=2,N,2
   X(I)=Y(I)+Z(I)
END
        .
        .
        .
```

In this example, X(I) will be calculated for all even values of I from 2 to N; then control will pass to the statement following the structure's END statement. Initialization, incrementing, and testing are implied by the structure and are not explicitly programmed.

The general form of the DO-FOR statement is

```
DO FOR I=N1,N2,N3
```

where I is the index and N1, N2, and N3 are the initial, terminal, and increment parameters, respectively. The index of a DO-FOR must be an integer variable and may not be redefined within the body of the DO-FOR structure. The SFTRAN precompiler does not check this; it is the programmer's responsibility to ensure that such redefinitions do not occur.

The DO-FOR parameters determine the initial value N1, the increment value N3, and the number of times (N2-N1)/N3+1 that the body of the DO-FOR structure will be executed. These parameters may be literal integer constants, integer variables, or integer expressions. A complicated example would be

```
DO FOR INDEX = 0, IFUNC(X)+J*K, -NUM
```

Variables appearing in N1, N2, and N3 may be changed during execution of the statements contained within the DO-FOR structure with no effect; only their values at the start are used to control indexing. If N3 is omitted, its value is assumed to be 1 (unless N1 and N2 are literal constants and the value of N1 is greater than that of N2, in which case N3 is assumed to be -1).

Recapitulating, the DO-FOR structure is executed as follows:

(1) The index is initialized to the value of N1.

(2) The values of N2 and N3 are saved.

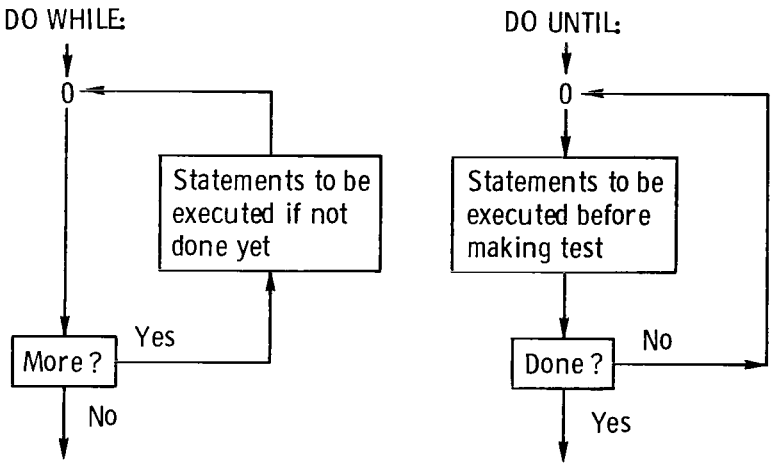
(3) The statements contained in the body of the DO-FOR structure are executed.

(4) The index is increased by the value of N3.

(5) If N3*(N2-I) is negative, the DO-FOR is completed. Otherwise, steps 3 to 5 are repeated.

When a DO-FOR structure is completed normally (by step 5), the value of the index is not the same as it was during execution of the body of the structure (in step 3) the last time.

DO-WHILE and DO-UNTIL. - In more general cases of looping, the block of statements must be repeated until a certain condition is attained, or while a certain condition holds. In SFTRAN these are accomplished by means of the DO-UNTIL and DO-WHILE structures, respectively. Their flow diagrams are



An example of DO-WHILE coding

```
        .
        .
        .

DO WHILE (N.GT.0)
   DO (PROCESS N-TH ITEM IN LIST)
   N=N-1
END

        .
        .
        .
```

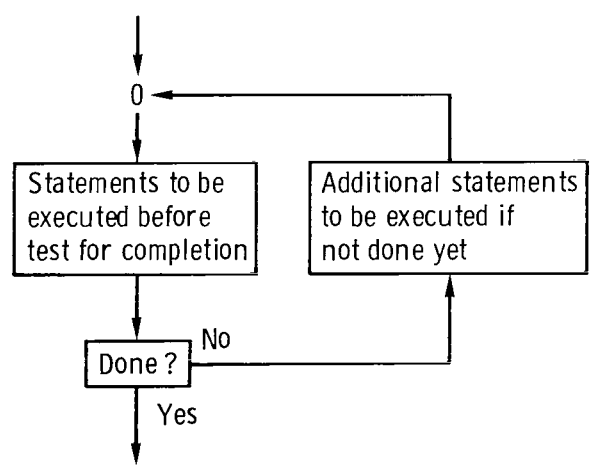
(In a DO-WHILE the logic test is made at the start, to determine if conditions for looping are allowed. Hence, if N should be 0 at the start, control will pass immediately to the statement following the structure's END statement.)

An example of DO-UNTIL coding is

```
        .
        .
        .
VALUE=GUESS
DO UNTIL (TERM.LE.SMALL)
   DO (CALCULATE HIGHER-ORDER TERM)
   VALUE=VALUE+TERM
END
        .
        .
        .
```

(In a DO-UNTIL the logic test is made at the end, to determine if looping is to continue. Hence, TERM, the first correction to GUESS, will be calculated and added before being examined to see whether it is SMALL enough to discontinue looping.) It is quite possible to program an infinite loop, in which the completion test is never satisfied. The SFTRAN precompiler cannot check this. It is the programmer's responsibility to avoid infinite loops.

DO-WITH. - In more general cases of looping it is desirable to have the completion test take place other than at the start or end of the loop. For these cases the DO-WITH structure is available, whose flow diagram is



An example of DO-WITH coding is

```
        .
        .
        .
ITEM=1
DO WITH
   DO (CHECK FOR MATCH WITH STANDARD VALUE)
UNTIL (MATCH.OR.ITEM.EQ.LAST)
   DO (PROCESS NON-MATCHING ITEM)
   ITEM=ITEM+1
END
IF (.NOT.MATCH) DO (REPORT MISSING ITEM)
        .
        .
        .
```

In this case, the DO-WITH completion test is indicated by the keyword UNTIL statement. A WHILE statement could also have been used. For the UNTIL test, a .TRUE. value signals completion of the looping. For the WHILE test, a .FALSE. value signals completion. In either case, one and only one WHILE/UNTIL statement may be used in the DO-WITH structure, but it may be placed anywhere within the body of the structure.

Additional Forms

The basic structures just defined are more than adequate to describe even the most complex program flow. But, because of its FORTRAN origin, SFTRAN has been given three additional forms: READ and WRITE parameters, EXITS from a DO-FOR, and INCLUDE and DEFINITION.

READ and WRITE parameters. - READ and WRITE statements in SFTRAN may contain END and/or ERR parameters, just as they do in FORTRAN. In SFTRAN, however, these parameters set logical variables instead of causing control transfers. For example, the statement

```
READ (UNIT,FMT,DONE=END) LIST
```

will read into LIST from UNIT according to the format specified in FMT. If an end-of-file is encountered, the logical variable DONE will be set to .TRUE.; otherwise, DONE will be set to .FALSE. The following example is part of a main program that makes use of this feature:

```
      .
      .
      .
DO WITH
   READ (5,100,DONE=END) DATA
UNTIL (DONE)
   DO (PROCESS ALL DATA IN THIS GROUP)
   DO (PRINT RESULTS OF PROCESSING)
END
STOP
      .
      .
      .
```

To illustrate the use of the ERR parameter, suppose that the READ statement in this example is replaced by

```
READ (5,100,DONE=END,SCREWY=ERR) DATA
IF (SCREWY) THEN
   KIND=4
   DO (ERROR HANDLING ROUTINE)
END
```

The program will operate just as before, unless an error is encountered at READ time. In that event, SCREWY will be set to .TRUE., and then the procedure ERROR HANDLING ROUTINE will be invoked for KIND=4. (If no READ error is encountered, SCREWY will be set to .FALSE.)

EXITS from a DO-FOR. - It has been stated that every SFTRAN structure has only one entrance and one exit. Like most generalities, this admits of an exception: the EXIT statement. The EXIT statement, which may be used only in connection with a DO-FOR structure, has been added to give that structure the feature of having a completion test within the body of the structure. In this respect it is similar to the UNTIL or WHILE statements of the DO-WITH structure. For instance, the statement

```
IF (EQUAL) EXIT
```

in a DO-FOR structure is roughly analogous to the statement

```
UNTIL (EQUAL)
```

in a DO-WITH structure. The EXIT statement, however, possesses a feature that makes it much more powerful than UNTIL/WHILE statements: it can optionally include the name of a DO-FOR structure index, in parentheses, to cause EXIT from that DO-FOR structure. In this respect, it furnishes a means of unconditional transfer out of a nest of structures.

Consider, for example, the following SFTRAN code:

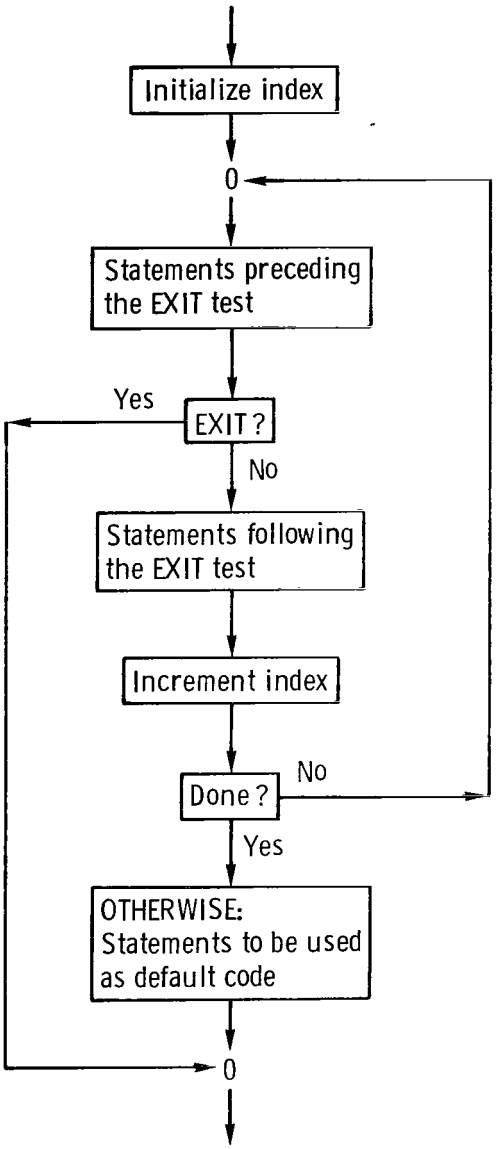
```
        .
        .
        .
DO FOR ITEM=1,LAST
   DO (GET DATA ITEM AND ALL 5 STANDARD VALUES)
   DO FOR KIND=1,5
      DO (CHECK FOR MATCH WITH THIS KIND OF STANDARD)
      IF (MATCH) THEN
         DO (PROCESS MATCHING CASE)
   <-----EXIT (KIND)
      END
      DO (PROCESS NON-MATCHING CASE)
   END
END
        .
        .
        .
```

When a MATCH is found, it is processed and then control is transferred out of the IF-THEN structure to the first statement following the END statement of the DO-FOR structure with index name KIND. For clarity, this is indicated in the output listing (but not the source) by a left-going arrow. (The alternative structure

```
      IF (MATCH) THEN
         DO (PROCESS MATCHING CASE)
   <-----EXIT (KIND)
      ELSE
         DO (PROCESS NON-MATCHING CASE)
      END
```

is nominally correct, but its ELSE block is clearly unnecessary. Consequently, this form is not allowed, and the precompiler is programmed to regard it as an error. See the section Comments.)

In any event, it is obvious that the EXIT statement will be associated with a test of some sort. The question then arises, What about the possibility that no EXIT has occurred by the time the loop is finished? Without a special provision, control would pass to the next statement following the appropriate END statement - just as if an EXIT had occurred. To handle this situation, SFTRAN provides the keyword OTHERWISE (again, to be used only in the DO-FOR structure). The flow diagram for a DO-FOR with an EXIT and an OTHERWISE is



An example of DO-FOR coding with an EXIT and an OTHERWISE is

```
    .
    .
    .
DO FOR KIND=1,5
   DO (CHECK FOR MATCH WITH THIS KIND OF STANDARD)
<--IF (MATCH) EXIT
   DO (PROCESS NON-MATCHING CASE)
OTHERWISE
   DO (REPORT NO MATCH FOUND)
END
    .
    .
    .
```

In this case, the procedure REPORT NO MATCH FOUND will be executed if the item matches none of the standards. Normally, however, a match will be found, and control will be transferred out of the DO-FOR structure. (The optional, abbreviated version of EXIT has been used that omits THEN and END, and the index name KIND has been omitted because exit is from the immediate structure.)

INCLUDE and DEFINITION. - FORTRAN compilers often require that certain classes of statements appear in the program before certain other classes of statements. (For example, function definition statements must appear before any executable statements.) In SFTRAN, the INCLUDE statement provides a way to specify where a block of statements, defined elsewhere in a DEFINITION structure, is to be located in the FORTRAN output from the precompiler.

The INCLUDE statement allows the programmer to indicate the presence of a group of statements without spelling them out. Thus, he can express main ideas at the top of the program without the clutter of detail, best left until later. As with DO-PROCEDURE statements and PROCEDURE blocks, INCLUDE statements are related to their corresponding DEFINITION blocks by a unique (hopefully descriptive) name. An example of the INCLUDE statement is

```
            .
            .
            .
      INCLUDE (TYPE STATEMENTS, ETC.)

C.....MAIN FLOW:
      DO (INITIALIZE PROGRAM)
      DO WITH
         READ (5,INPUT,DONE=END)
      UNTIL (DONE)
         DO (FIRST-ORDER CALCULATION)
            .
            .
            .
```

Then, at another point in the program, perhaps at the bottom, would be

```
            .
            .
            .
      DEFINITION (TYPE STATEMENTS, ETC.)
         LOGICAL DONE
         REAL X(25),Y(25)
         NAMELIST /INPUT/ X,Y,NXY
      END
            .
            .
            .
```

In the FORTRAN output from the precompiler, the statements LOGICAL --, REAL --, and NAMELIST -- will appear in place of the INCLUDE statement, and the DEFINITION and END statements will not appear.

There are a few rules governing the use of INCLUDE and DEFINITION statements. DEFINITION statements must stand alone; they cannot be contained in any other structure. INCLUDE statements must not be contained in DEFINITION blocks (because they cannot be nested). The names of INCLUDE and DEFINITION statements should not contain apostrophes or unmatched (left with right) parentheses. And names must match exactly; each name must appear only twice, once in an INCLUDE statement and once in a DEFINITION statement.

Comments

As noted, the EXIT statement requires special attention because it can cause unconditional transfer out of a structure (other than at its normal END). There are also two common, allowable FORTRAN statements that possess this characteristic: RETURN and STOP. The rules governing the use of EXIT, RETURN, and STOP within structures are as follows:

(1) They may be used in IF-statements.

(2) They may be used alone as the last statement in an IF-THEN structure, an OTHERWISE block of a DO-FOR structure, or a CASE or IF-NONE block of a DO-CASE structure.

(3) No other use within structures is allowed.

Occasionally, errors in form or spelling may cause the SFTRAN precompiler to pass over statements (thinking them to be FORTRAN) when the programmer had intended them as SFTRAN. The FORTRAN compiler will, of course, complain about these, and the programmer should then be able to determine what caused the precompiler to ignore the statements in the first place.

Normally, nonexecutable statements are placed at the start of a program, either directly or by means of an INCLUDE statement. When they are not, the SFTRAN precompiler may assign a statement number to one or more of them. This is reported as an error by the compiler. (Most FORTRAN compilers do not allow labels on nonexecutable statements.) One way to fix this is to insert a CONTINUE statement just ahead of any nonexecutable statements that were labeled and to precompile again.

Some changes in an SFTRAN code may be required when changing from the FORTRAN V compiler to the ASCII FORTRAN compiler, or conversely, because of the differences between these FORTRAN dialects. For example, COMPLEX*16 and CHARACTER variable types are permitted in ASCII FORTRAN but not in FORTRAN V; characters are stored six to a word in FORTRAN V but four to a word in ASCII FORTRAN.

These and other differences between the two languages are more completely described in the ASCII FORTRAN programmer manual (ref. 1).

Finally, SFTRAN does not permit free-form coding. Columns 1 to 5 are reserved for statement numbers (except in comment lines), and column 6 is reserved for a continuation character. (Hyphens should not be used to indicate that a statement is continued on the following line.) Statement scanning begins in column 7; any initial blanks will be regarded as relative indenting and will be preserved by the precompiler. The SFTRAN precompiler uses only the first 72 columns of each line.

TASK MANAGEMENT SYSTEM

For the following discussion, it is essential that the reader be familiar with the UNIVAC 1100 series Conversational Time Sharing System (CTS) and also its programmer's reference manual (ref. 2). The command subroutines and file organization described here supplement the commands that are a part of CTS; they do not replace them. The task-management commands are summarized in appendix A. These new commands provide the following features:

(1) They greatly simplify the management of several jobs under one PROJECT ID and at the same time make it easy for several programmers to work as a team on one job.

(2) They provide all of the library functions essential in multijob, multiprogrammer environments without the need to assign this work to an individual.

(3) They make full use of, and are fully compatible with, UNIVAC's CTS facilities. Only standard features of task and data management are required.

(4) They provide features that are equally available in either conversational or batch modes.

It is recommended that a single program file be used for source code, for relocatable code, and for listings. The command subroutines, which assume this organization, are provided to simplify task management.

To illustrate the method, suppose that the assignment is to analyze sonic booms and that the programmer has written a main program (MAIN) and two subroutines (INPUT and OUTPUT), all in SFTRAN. The first step is to put MAIN, INPUT, and OUTPUT into the program file, called (say) SONIC-BOOM (or any other name the programmer likes). That is, SONIC-BOOM is created with elements MAIN, INPUT, and OUTPUT. The UNIVAC 1100 program-file organization provides an excellent filing scheme. The next step is to make relocatable elements corresponding to MAIN, INPUT, and OUTPUT. Suppose the programmer begins with MAIN, invoking the SFTRAN precompiler. This would produce two new file elements, one for further processing and one for listing purposes.

The first element, called TPF$.MAIN, is the FORTRAN version of MAIN and is ready for processing by the FORTRAN compiler. After compilation, the resulting relocatable code will be put into the program file as relocatable element MAIN. (The UNIVAC 1100 system file-naming conventions permit both source element MAIN and relocatable element MAIN to be in the same file.)

The second element produced by the precompiler has been put into the program file as element MAIN$L. The latter is an SFTRAN version of MAIN that has been indented to reveal how the structures are nested. (In SFTRAN, statement text is assumed to begin in column 7, except for comment statements. If leading blanks do occur beginning in column 7, these will be preserved by the precompiler as additional indentation, relative to that showing structure nesting.)

Occasionally, the need arises to save a FORTRAN source element for a particular job. In the present case, for example, a Bessel function subroutine like BESORD might be required and could be stored in SONIC-BOOM as element BESORD$F. The $F suffix on BESORD$F would identify it as a FORTRAN-language element.

Command Subroutines

A number of command subroutines have been provided to assist in creating and managing the SFTRAN program file:

JOB	EDITLIB	EDPRINT
SFTRAN	ERASELIB	PARAM
FORTRAN	LISTINGS	DEFAULT
COLLECT	PRINTLIB	

(They have been grouped according to their functions: processing, editing and listing, and miscellaneous.)

Task selection is accomplished by command JOB; relocatable elements are created either from SFTRAN-language program-file elements, by using command SFTRAN, or from FORTRAN-language program-file elements, by using command FORTRAN. Relocatable elements are collected to build an absolute element by using command COLLECT. Elements of the program file may be created and edited by command EDITLIB; when a given program is no longer needed, source, relocatable, and listing elements are removed from the program file by command ERASELIB. Batch prints of indented source-code listings produced by the SFTRAN precompiler are obtained by command LISTINGS; batch prints of original, unprocessed program-file elements may be obtained by command PRINTLIB.

Batch prints of a file or file element are ordered by command EDPRINT if the first character of each record is to be used for printer control. The PARAM command subroutine implements keyword and positional operand evaluation for the SFTRAN command subroutines described and can also be used in user-developed command subroutines. Default values for operand keywords are set by the command DEFAULT.

In this section, examples are given to show how the command subroutines are called. A summary of the commands and their operands is given in appendix A.

JOB. - At the start of each task creation and management session, one normally issues a command such as

```
CALL JOB SONIC-BOOM
```

This sets up file assignments and default values so that subsequent commands refer to the correct file elements. In addition to defining the program file, the JOB command may also be used to establish standard operating procedures. For example, one might enter

```
CALL JOB SONIC-BOOM,  NUMBER=Y,STRIP=Y,OFFLINE=Y
```

These additional operands require that (unless otherwise specified, when command subroutine SFTRAN is called)

(1) The various elements of SFTRAN structures be numbered on all SFTRAN listings
(2) Programmer-supplied statement numbers be stripped from all SFTRAN listings
(3) Error messages and their line numbers go offline to the SFTRAN listings, not to the terminal

(Normally, the internally generated statement numbers are not shown, programmer-supplied statement numbers are not stripped, and error messages appear at the terminal.)

Another version of the JOB command might be

```
CALL JOB SONIC-BOOM, FORTRAN=N,LISTING=Y,LINES=Y
```

These additional operands require that (unless otherwise specified, when command subroutine SFTRAN or command subroutine FORTRAN is called)

(1) FORTRAN source data sets created by the precompiler are not to be compiled
(2) A listing of indented SFTRAN output (or a FORTRAN listing in the event that command subroutine FORTRAN is used) is to be printed for each member processed by the SFTRAN command subroutine
(3) Line numbers are to be included in listings

(Normally, the FORTRAN compiler will be called if no precompiler errors are detected, SFTRAN output listings are not printed since they may not be the final version desired, and listings are normally printed without line numbers.)

Any combination of these operands may be used when JOB is invoked. The choices

specified then hold for all subsequent processing during that terminal session, unless revoked by another JOB command (or temporarily overridden by a particular SFTRAN or FORTRAN call).

Another example of the JOB command might be

```
CALL JOB SONIC-BOOM,  COMPILER='FTN,SO'
```

The COMPILER operand allows the user to specify the FORTRAN compiler and its options that will be used in subsequent SFTRAN or FORTRAN commands. (Unless specified, the assumed compiler is 'FTN,SF'; the ASCII FORTRAN compiler with option S for source-code and error-message listing and option F for postmortem walk-back and dump capability will be used.) Quotes are required around the COMPILER operand value because of the comma in the value string.

At times it may be desirable to limit access to a program file by attaching a READKEY, which limits read access to those who know the key, or a WRITEKEY, which limits write access. The JOB command has operands to accommodate these keys. For example, suppose that a new program file, MY-LIB, is to be created. The command

```
CALL JOB MY-LIB,  WRITEKEY=XYZ
```

will create the file MY-LIB with WRITEKEY 'XYZ'. Once this file has been freed (by calling JOB again, or when the task is terminated), MY-LIB will be write-protected. That is, it will not be possible to alter or erase MY-LIB without first giving the WRITEKEY combination 'XYZ'. If MY-LIB already exists when the JOB command is issued, the WRITEKEY value is compared with that on MY-LIB for validity and, if correct, the user is enabled to update the file. In this way, one can conveniently obtain protection against accidental destruction of a program file or any of its elements. The READKEY operand is used similarly to obtain file privacy.

SFTRAN. - The command SFTRAN is used to process source elements, written in SFTRAN, that are stored in a particular program file. First, each element is precompiled, creating SFTRAN output for listing and a FORTRAN source element; then (depending on defaults) the SFTRAN output is printed and the FORTRAN source element is compiled.

Suppose, for example, that the JOB name is SONIC-BOOM, the JOB command has been issued previously, and there are two elements to process. These are stored in the program file as elements MAIN and SEARCH. Then, to begin processing, enter

```
CALL SFTRAN MAIN,SEARCH
```

If no precompiler errors or compiler errors are detected, the user will eventually be prompted for more commands. When this has happened, the following elements will have been created:

```
SONIC-BOOM.MAIN$L
SONIC-BOOM.SEARCH$L
TPF$.MAIN
TPF$.SEARCH
```

The first two are indented SFTRAN output produced by the precompiler and may be printed using LISTINGS. The second two are FORTRAN source elements, in temporary program file TPF$, which are used by the FORTRAN compiler to create relocatable elements MAIN and SEARCH in the program file.

If the precompiler detects errors, messages will appear at the terminal (unless OFFLINE=Y is specified when the JOB or SFTRAN command is issued) and the FORTRAN compiler will not be invoked. If the FORTRAN compiler detects errors, the number of errors will be reported. Details of these errors can be obtained by using the CTS editor to examine the working area *f*, which at this point contains the FORTRAN compilation listing.

Other forms of the SFTRAN command are possible, such as

```
CALL SFTRAN MAIN,SEARCH, NUMBER=Y,STRIP=Y,OFFLINE=Y
```

or

```
CALL SFTRAN MAIN,SEARCH, LISTING=Y,FORTRAN=N
```

or

```
CALL SFTRAN MAIN, COMPILER='FTN,SO'
```

The keyword operands in these examples have been discussed, and their use here is to override (temporarily) the choice made with the JOB command.

FORTRAN. - The command FORTRAN is used to compile elements, written in FORTRAN, that are stored in a particular program file. Such elements may have already been precompiled and saved (unlikely), or they may be special programs not intended for the precompiler (e.g., obtained elsewhere in FORTRAN).

Consider, for example, the FORTRAN subroutine BESORD. This would probably be stored in the source library as element BESORD$F, with the $F suffix indicating that it is written in FORTRAN and is not suitable for the precompiler. (Actually, the precompiler would simply output each line of BESORD$F as it received it, without modification.) Then, assuming that the JOB command has been previously issued, one enters

```
CALL FORTRAN BESORD$F
```

After the prompt is received, the relocatable element BESORD$F will exist in the program file.

The command FORTRAN also has the operand keywords COMPILER, LISTING, and LINES, which may be used to override (temporarily) the choice made with the JOB command.

COLLECT. - The command COLLECT is used to build an absolute element ready for execution. Assuming that the JOB command has been issued and that the main program is called MAIN, enter

```
CALL COLLECT MAIN
```

to produce absolute element MAIN in the temporary program file TPF$. The collection is done with SONIC-BOOM. MAIN used as the main-program relocatable element; subprograms referenced by MAIN will be taken from SONIC-BOOM if they exist. (The COLLECT subroutine includes a statement that causes the library UNIVAC*AFORLIJLIB to be searched in order to satisfy unresolved references. The user must remove this statement if the FORTRAN V compiler was used to create relocatable elements in the program file.)

Other forms of the COLLECT command are possible, such as

```
CALL COLLECT MAIN,COM1,COM2, JOBLIB=MY-LIB
```

which would include block-data subprograms COM1 and COM2 from SONIC-BOOM and would use the program file MY-LIB as an additional JOBLIB. If more than one JOBLIB name is specified (by using quotes around a string of names separated by commas), these will be searched in the order given. The absolute program is still known as element MAIN because MAIN is the first parameter given.

Another form might be

```
CALL COLLECT MAIN, INCLUDE=MY-LIB.INPUT,SAVEABS=Y
```

which would use relocatable element INPUT from MY-LIB instead of from SONIC-BOOM and would save the absolute program by copying it from TPF$.MAIN to SONIC-BOOM. MAIN. (The user is cautioned that the COLLECT command erases the contents of TPF$ before building the new absolute element there. This is done to prevent inclusion of relocatable elements that might previously have been put into TPF$.)

EDITLIB. - The command EDITLIB is used to create or edit an element of a particular program file by using the CTS editor. Suppose the element name is, or is to be SEARCH; then, assuming the JOB command has been issued, enter

```
CALL EDITLIB SEARCH
```

The usual prompts and messages from the CTS editor will appear. When editing is completed, the results may be preserved by entering SAVE or REPLACE, as appropriate.

ERASELIB. - The command ERASELIB is used to remove elements that are no longer wanted from the program file. For example, assuming that the JOB command has been issued, enter

```
CALL ERASELIB SEARCH,INPUT
```

to remove any elements that were produced by issuing the commands EDITLIB, SFTRAN, FORTRAN, or COLLECT with the element names SEARCH or INPUT.

LISTINGS. - The command LISTINGS is used to obtain listings of indented SFTRAN code produced by the precompiler. It assumes that the element already exists in the program file. (If this is not the case, perhaps because of an error in typing the element name, the user will be informed that an element was not found.) Suppose that prints of the SFTRAN listings of MAIN and SEARCH are desired; then, assuming the JOB command has been issued, enter

```
CALL LISTINGS MAIN,SEARCH
```

This will create a SYMBIONT file for batch printing with a unique name that is decataloged when the print has been completed. This single file will contain listings of both MAIN and SEARCH, with identifying headings that will appear on each printed page. Note that the listing elements printed are actually MAIN$L and SEARCH$L but that the $L suffixes are not included in the LISTINGS command.

As another example, the command

```
CALL LISTINGS OUTPUT, LINES=Y
```

will produce a listing of the SFTRAN-language element OUTPUT with line numbers.

PRINTLIB. - The command PRINTLIB is used to obtain batch prints of source-code elements in the program file. These will be original (nonindented) SFTRAN codes, certain FORTRAN source codes that have been stored, or documentation describing the job and the programs. For example, assuming that the JOB command has been issued, enter

```
CALL PRINTLIB MAIN,BESORD$F,SYSDOC
```

to obtain batch prints of MAIN, BESORD$F, and program documentation SYSDOC. This will create a temporary SYMBIONT print file that will be printed on the high-speed printer. This single file will contain copies of MAIN, BESORD$F, and SYSDOC, with identifying headings that will appear on each printed page. As another example, the command

```
CALL PRINTLIB INPUT, LINES=Y
```

will produce a listing of the element INPUT with line numbers.

EDPRINT. - The command EDPRINT is used to print a source-language file or element on the high-speed printer with the first byte (column 1) of each line used for printer control. For example,

```
CALL EDPRINT 8.
```

could be used to print the temporary file produced by a FORTRAN program that writes on UNIT 8. (However, the EDPRINT command will not work for SYMBIONT files that are

produced by using the BRKPT command of the UNIVAC 1100 EXEC system to divert run-stream output from PRINT $ to a user file.

PARAM. - The command PARAM was created specifically to implement keyword and/or positional operand evaluation in the CTS command subroutines described. It may also be used in user-developed command subroutines. For purposes of illustration, consider the following command subroutine that implements the command DEMO:

```
100  CALL PARAM DEMO:VALU1,VALU2,VALU3
110  TYPE 'VALU1=%VALU1%'
120  TYPE 'VALU2=%VALU2%'
130  TYPE 'VALU3=%VALU3%'
140  RETURN
```

(DEMO is not one of the SFTRAN task-management commands; it is included here only to illustrate use of the PARAM command.) Suppose subroutine DEMO is executed by entering

```
CALL DEMO 1,2,3
```

When line 100 of DEMO is executed, the values of the variables VALU1, VALU2, and VALU3 will be set to 1, 2, and 3, respectively, by the PARAM command. Then lines 110, 120, and 130 will cause

```
VALU1=1
VALU2=2
VALU3=3
```

to be output on the terminal. Notice that the name of the calling subroutine, DEMO, must appear immediately before the colon in the call to PARAM.

As another example, one might enter

```
CALL DEMO VALU2=ALPHA,VALU3=BETA
```

In this case the response will be

```
VALU1=
VALU2=ALPHA
VALU3=BETA
```

Since no value was given for VALU1 (either by position or by keyword), the default value is used - in this case a null string. The user is encouraged to try this little command subroutine with other operand values to become more acquainted with the PARAM command.

Successful use of the PARAM command requires that the keyword names chosen be alphanumeric, that they begin with a letter, and that they be no longer than eight characters.

DEFAULT. - The command DEFAULT is used to set default values for operand keywords. For example (continuing from the preceding discussion), if one enters

```
CALL DEFAULT VALU1=14,VALU3='ABC,DEF'
CALL DEMO VALU2=ALPHA
```

the system will respond with

```
VALU1=14
VALU2=ALPHA
VALU3=ABC,DEF
```

That is, the defaulted values for VALU1 and VALU3 are used because they were not supplied in the DEMO command. Default values so specified remain in effect during the entire terminal session unless they are changed by subsequent DEFAULT commands with the same keywords. Users are urged to try other combinations of the DEMO and DEFAULT commands to become better acquainted with the use of the DEFAULT and PARAM command subroutines.

Comments Concerning Task Management

A final comment about the command subroutines may be useful. Some of them can be used recursively; that is, up to 10 element names may be entered, in addition to the keyword operands. Those that cannot are JOB, PARAM, and DEFAULT (which take no member name operand), EDITLIB (for which recursion would be of doubtful value), and EDPRINT (which can be used to print only one file or element at a time). The DEFAULT command may include more than one keyword=value field.

Nine other command subroutines are included in the SFTRAN package:

BLOCKHDR	GENTIME0	QUEUEX
FINDNAME	GENTIME	REQUEUEX
FINDVALUE	NAMEMATCH	READFILE

These commands are not designed for direct call by the user. They are called by the command subroutines already described and hence are a necessary part of the whole SFTRAN package.

ACQUIRING SFTRAN

The SFTRAN precompiler absolute program, two programs for creating SYMBIONT files for batch prints, and the task-management command subroutines are contained in a program file name SFT1100. On the Lewis Research Center's UNIVAC 1100 system, this file has the qualifier FESSLER and read access is permitted to all users.

To acquire SFTRAN capability, the user should issue the CTS command

```
COPY,S FESSLER*SFT1100.,
```

The result will be that the user's CTS program file, F, will receive copies of each of the SFTRAN command subroutines.

A brief runstream with a simple SFTRAN program is provided in appendix B to help new users get started. Listings of a larger SFTRAN program are provided in appendix C to illustrate good use of the language features.

SFTRAN PRECOMPILER

The precompiler translates SFTRAN source code to FORTRAN source code. It was written in SFTRAN and consists of a main program, a BLOCK DATA subprogram, and nine other subprograms (excluding system-supplied routines). Communication between these is partly by calling arguments and partly by the two common blocks SFTSET and SFTCOM.

Precompiler Main Program

All parameters that control the operation of the precompiler are contained in the common block SFTSET. These are set at the start of each precompilation by the command subroutine SFTRAN, which adds appropriate NAMELIST input lines to the runstream.

Statement analysis and translation is performed in two passes in the main program. In pass 1, SFTRAN statements in the user's program are recognized and translated to FORTRAN statements. The precompiler is transparent to non-SFTRAN statements and these pass through without change. If errors are detected, they are reported along with the offending SFTRAN statement. Syntax errors are rarely fatal; pass 1 is nearly always completed. An indented listing of the user's SFTRAN program is also produced in pass 1.

A feature of SFTRAN is the use of descriptive names to refer to a block of statements. At the conclusion of pass 1, a check is made to see that the name of each PROCEDURE block was used at least once in a DO-PROCEDURE statement, and conversely. Also, a check is made to see that the name of each DEFINITION block was used in an INCLUDE statement, and conversely. Then, if no errors have been detected up to this point, pass 2 begins.

Pass 2 operates on the translated, all-FORTRAN output from pass 1. Two files are read by pass 2: one contains the bulk of the pass 1 translation and the other contains those statements that came from DEFINITION blocks in the user's SFTRAN program. In pass 2, these two files are remerged; INCLUDE statements (which are not translated in pass 1) are replaced by those statements that came from the DEFINITION block of the

same name. Another function of pass 2 is to append a list of statement numbers to the ASSIGNED GO TO statements generated in pass 1 at the end of each PROCEDURE block.

Precompiler Subprograms

INOUT. - All statement input and output occurs in subroutine INOUT, which has four entry points. Entry point INPUT is for input of both SFTRAN and FORTRAN statements. Output of SFTRAN statements is by entry point SFOUT, output of FORTRAN statements is by entry point F4OUT, and output of error statements is by entry point ERROUT. Statement line numbers, concatenation of input lines in the case of multiline statements, and generation of continuation lines on output are taken care of in INOUT. In this way, system-dependent details are confined to one subprogram.

SCAN. - The statement scanning functions FIND and LOCATE are done in logical-function subprogram SCAN. FIND tests whether or not a given string next occurs in the current statement. LOCATE searches the current statement, starting with the current character, for a specified character. SCAN is also an entry point name; this entry causes scanning to begin (or resume) at a particular byte (card column) of the current statement.

An important part of SCAN is a test to see if there are any more nonblank characters in the current statement beyond the current string. This test is performed at each call to SCAN and whenever a successful call to FIND or LOCATE occurs. If more characters are found, the position and value of the first one are obtained; if only blank characters (spaces) remain, these are eliminated by a reduction of the statement-length count.

NCODE and DCODE. - Subprograms NCODE and DCODE convert internal integer numbers to digit strings, and conversely. When a character string is decoded, the value of logical function DCODE is .FALSE. if any decoding errors were encountered. (One of the uses of DCODE is to determine whether or not a string of characters represents a literal integer constant.)

ERROR. - Subroutine ERROR has two entry points: ERROR, for nonfatal error messages, and HALT, for fatal error messages. In this subroutine, the error message is received in the argument string as a parenthetical expression. These parentheses are located and the error message is extracted and appended to *ERROR*, or *FATAL ERROR*, as the case may be. This complete message is then announced, together with the current SFTRAN statement responsible for the message, by calls to ERROUT.

PARENS. - Logical-function subprogram PARENS scans a string to see if a complete parenthetical expression begins at a specified byte (or is preceded by only blank characters). If left and right parentheses are found, their byte positions are returned.

ADDBUF. - Subroutine ADDBUF adds a string of characters to the output buffer used for generating FORTRAN statements and increments the count of the total number of characters in the buffer.

Character-Manipulating Routines

Two character-manipulating subprograms, which are not part of the FORTRAN library, are used in the SFTRAN precompiler. These give SFTRAN and FORTRAN programs the ability to deal with imbedded strings of characters.

F4MVC(A,I,B,J,N). - Subroutine F4MVC moves N characters from string A to string B, starting with the I^{th} character of A, which replaces the J^{th} character of B. I, J, and N must all be greater than zero.

F4CLC(A,I,B,J,N). - Logical-function subprogram F4CLC compares N characters of string A with N characters of string B, starting with the I^{th} character of A, which is compared to the J^{th} character of B, etc. I, J, and N must all be greater than zero. A .TRUE. value is returned only when a match is obtained for all N characters.

Lewis Research Center,
National Aeronautics and Space Administration,
Cleveland, Ohio, January 24, 1978,
505-01.

APPENDIX A

SUMMARY OF TASK-MANAGEMENT COMMANDS

In summarizing the task-management commands, the following notation is used: command names, keywords, and formal operands are in upper case and functional operands are in lower case. Also, several metasymbols are used:

- [ ] delimits optional operand fields
- { } delimits operand alternatives
- | separates operand alternatives
- . . . indicates that the preceding field may be repeated (up to 9 times)

Command	Operands
COLLECT	element name[,...] [,JOBLIB=file name(s) to be used as a library] [,INCLUDE=element(s) from other files to be included] [,SAVEABS={Y|N}]
DEFAULT	[keyword=value] [,...]
EDITLIB	element name
EDPRINT	{file name}|{file and element name}
ERASELIB	element name[,...]
FORTRAN	element name[,...] [,COMPILER='compiler name, options'] [,LISTING={Y|N}] [,LINES/{Y|N}]
JOB	[job name] [,FORTRAN={Y|N}] [,NUMBER={Y|N}] [,STRIP={Y|N}] [,OFFLINE={Y|N}] [,LISTING={Y|N}] [,LINES={Y|N}] [,COMPILER='compiler name, options'] [,READKEY=read-key string] [,WRITEKEY=write-key string]
LISTINGS	element name[,...] [,LINES={Y|N}]
PARAM	name of calling subroutine: [KEYWORD] [,KEYWORD],...
PRINTLIB	element name[,...] [,LINES={Y|N}]
SFTRAN	element name[,...] [,FORTRAN={Y|N}] [,NUMBER={Y|N}] [,STRIP={Y|N}] [,OFFLINE={Y|N}] [,LISTING={Y|N}] [,LINES={Y|N}] [,COMPILER='compiler name, options']

APPENDIX B

SAMPLE RUNSTREAM

The following runstream is an example of how the SFTRAN system is used at a terminal. The usual LOGON procedure and the procedure described previously for acquiring SFTRAN are assumed to have already been completed. User-entered lines are displayed in lower case and system responses are displayed in upper case.

```
->call job learn
NEW FILE LEARN ASSIGNED
->call editlib tryl
NEW ELEMENT TRYl OPENED
->number 100,100 n
>c.....this is my first try at sftran programming.
>       include (preliminaries)
>c
>       do with
>       read (5,1,done=end) string
>       until (done .or. string(1).eq.blank)
>       write (6,2) string
>       end
>       stop
>c
>   1   format (10a4)
>   2   format ('  you said: ',10a4)
>       definition (preliminaries)
>       logical done
>       dimension string(10)
>       data blank /' '/
>       end
>       end
>*save
->call sftran tryl
BEGINNING TRYl
FURPUR 27R2  RL72Rl 04/07/77 13:48:12
FURPUR 27R2  RL72Rl 04/07/77 13:48:20
COMPILING...
FURPUR 27R2  RL72Rl 04/07/77 13:48:47
 1 REL
20 END FTN 22 IBANK 34 DBANK
->call collect tryl,saveabs=y
IN EXEC MODE
FURPUR 27R2  RL72Rl 04/07/77 13:52:35
 1 ABS
->xqt learn.tryl
>123456
 YOU SAID 123456
>abc ... xyz
 YOU SAID ABC ... XYZ
 >
->call listings tryl
```

The indented listing of SFTRAN code produced by the last user command is as follows:

```
        LEARN.TRY1$L   07 APR 77    13:48:13

C.....THIS IS MY FIRST TRY AT SFTRAN PROGRAMMING.
      INCLUDE (PRELIMINARIES)
C
      DO WITH
         READ (5,1,DONE=END) STRING
      UNTIL (DONE .OR. STRING(1).EQ.BLANK)
         WRITE (6,2) STRING
      END
      STOP
C
  1   FORMAT (10A4)
  2   FORMAT ('  YOU SAID: ',10A4)
      DEFINITION (PRELIMINARIES)
         LOGICAL DONE
         DIMENSION STRING(10)
         DATA BLANK /' '/
      END
      END
```

APPENDIX C

SFTRAN EXAMPLES

The following listings are intended to illustrate the use of SFTRAN coding. The first example is an indented SFTRAN program listing as produced by LISTINGS. It contains all of the SFTRAN structures currently available and is in good "top-down" form. Next is a portion of the same demonstration program as it would appear when NUMBER=Y is specified. The last example is the FORTRAN code generated by the precompiler at the same time it produced the numbered SFTRAN listing.

The numbered examples were produced with LINES=Y to illustrate the correspondence between SFTRAN output and FORTRAN output numbering. Not shown is the degree to which these correspond to the input line numbers. The general rule can be stated quite simply: the SFTRAN precompiler makes every effort to give output lines index numbers that are 100 times the input line number; when additional output lines must be generated, their numbers are incremented by 1 (cf. lines 10800-10804 of the example FORTRAN listing). The result is that, although generally the programmer does not see his line numbers, if an error message refers to some specific line number, he knows exactly where (in the input element) it can be found.

One exception to the rule just mentioned is found in lines that have been moved as a result of the use of INCLUDE-DEFINITION statements. The starting index number of a moved block of statements will be 100 times the line number of the INCLUDE statement they replace.

Example 1

```
        UNIVACSFTRAN.DEMO$L   14 MAR 77 22:12:00   PAGE 1 OF 4

C.....PROGRAM TO DEMONSTRATE SFTRAN CODING.

C       THIS PROGRAM CALCULATES MOLECULAR WEIGHT FOR A
C       GIVEN MOLECULAR FORMULA.

        INCLUDE (TYPE AND DATA STATEMENTS)


C.....MAIN FLOW:

        DO WITH
           DO (INITIALIZE FOR NEW FORMULA)
           READ (5,500,DONE=END) FORMLA
        UNTIL (DONE)
           DO UNTIL (ERROR .OR. TYPE.EQ.0)
              DO (IDENTIFY NEXT BYTE TO DETERMINE PROCESSING TYPE)
              DO CASE (TYPE,3)
              CASE 1
                 DO (PROCESS NEW ELEMENT)
              CASE 2
                 DO (BEGIN NEW RADICAL)
              CASE 3
                 DO (END CURRENT RADICAL)
              END
           END
           IF (.NOT.ERROR) THEN
              IF (LEVEL.EQ.0) THEN
                 WRITE (6,600) MOLWT
              ELSE
                 WRITE (6,601)
              END
           END
        END
        STOP


C.....MAIN PROCEDURES:

        PROCEDURE (IDENTIFY NEXT BYTE TO DETERMINE PROCESSING TYPE)
           DO (GET NEXT BYTE FROM FORMULA)
           TYPE=1
           IF (BYTE.EQ.LPAREN)  TYPE=2
           IF (BYTE.EQ.RPAREN)  TYPE=3
           IF (BYTE.EQ.SPACE)  TYPE=0
           NEXT=NEXT+1
        END

        PROCEDURE (PROCESS NEW ELEMENT)
           DO (ASSEMBLE ELEMENT SYMBOL)
           DO (FIND MATCHING ELEMENT IN TABLE)
           IF (FOUND) THEN
              DO (READ NUMBER OF ATOMS/RADICALS)
```

```
            IF (LEVEL.EQ.0) THEN
               MOLWT=MOLWT+FLOAT(N)*ATWT(ATNO)
            ELSE
               RADWT(LEVEL)=RADWT(LEVEL)+FLOAT(N)*ATWT(ATNO)
            END
         END
      END

      PROCEDURE (BEGIN NEW RADICAL)
         LEVEL=LEVEL+1
         IF (LEVEL.GT.LMAX) THEN
            ERROR=.TRUE.
            WRITE (6,602)
         ELSE
            RADWT(LEVEL)=0.0
         END
      END

      PROCEDURE (END CURRENT RADICAL)
         LEVEL=LEVEL-1
         IF (LEVEL.GE.0) THEN
            DO (READ NUMBER OF ATOMS/RADICALS)
            IF (LEVEL.GT.0) THEN
               RADWT(LEVEL)=RADWT(LEVEL)+FLOAT(N)*RADWT(LEVEL+1)
            ELSE
               MOLWT=MOLWT+FLOAT(N)*RADWT(1)
            END
         ELSE
            ERROR=.TRUE.
            WRITE (6,603)
         END
      END


C.....MORE DETAILS:

      PROCEDURE (INITIALIZE FOR NEW FORMULA)
         MOLWT=0.0
         LEVEL=0
         NEXT=1
         ERROR=.FALSE.
      END

      PROCEDURE (ASSEMBLE ELEMENT SYMBOL)
         DO (PUT FIRST BYTE INTO SYMBOL)
         DO (GET NEXT BYTE FROM FORMULA)
         IF (BYTE.GE.SMALLA .AND. BYTE.LE.SMALLZ) THEN
            NEXT=NEXT+1
         ELSE
            BYTE=SPACE
         END
         DO (PUT SECOND BYTE INTO SYMBOL)
```

```
      END

      PROCEDURE (FIND MATCHING ELEMENT IN TABLE)
         DO FOR ATNO=1,NELEMS
            FOUND=SYMBOL.EQ.ELEMNT(ATNO)
         <--IF (FOUND) EXIT (ATNO)
         OTHERWISE
            ERROR=.TRUE.
            WRITE (6,604) SYMBOL
         END
      END

      PROCEDURE (READ NUMBER OF ATOMS/RADICALS)
         N=0
         DO WITH
            DO (GET NEXT BYTE FROM FORMULA)
         WHILE (BYTE.GE.ZERO .AND. BYTE.LE.NINE)
            N=10*N+(BYTE-ZERO)
            NEXT=NEXT+1
         END
         N=MAX0(N,1)
      END

C.....NOTE -- F4MVC (A,I,B,J,N) IS AN EXTERNAL SUBROUTINE WHICH MOVES
C             N CHARACTERS FROM STRING A TO STRING B, STARTING WITH
C             THE I-TH BYTE OF A WHICH REPLACES THE J-TH BYTE OF B.

      PROCEDURE (GET NEXT BYTE FROM FORMULA)
         CALL F4MVC (FORMLA,NEXT,BYTE,4,1)
      END

       PROCEDURE (PUT FIRST BYTE INTO SYMBOL)
         CALL F4MVC (BYTE,4,SYMBOL,1,1)
      END

      PROCEDURE (PUT SECOND BYTE INTO SYMBOL)
         CALL F4MVC (BYTE,4,SYMBOL,2,1)
      END


C.....MISCELLANEOUS:

500   FORMAT (19A4,A1)
600   FORMAT (' MOLECULAR WEIGHT =',F10.3)
601   FORMAT (' ERROR: PARENTHESES DO NOT MATCH')
602   FORMAT (' ERROR: TOO MANY NESTED RADICALS')
603   FORMAT (' ERROR: TOO MANY RIGHT PARENTHESES')
604   FORMAT (' ERROR: UNKNOWN ELEMENT = ',A2)

      DEFINITION (TYPE AND DATA STATEMENTS)
         IMPLICIT INTEGER (A-Z)
         REAL ATWT,MOLWT,RADWT
```

```
      LOGICAL DONE,ERROR,FOUND
      DIMENSION ATWT(110),ELEMNT(110),FORMLA(20),RADWT(10)

      DATA SPACE,LPAREN,RPAREN,SMALLA,SMALLZ,ZERO,NINE
     *     / 32,    40,    41,    97,   122,  48,  57/

      DATA LMAX,NELEMS,FORMLA,SYMBOL/10,20,21*' '/

      DATA ELEMNT(1),  ATWT(1)  / 'H',    1.00797 /,
     *     ELEMNT(2),  ATWT(2)  / 'He',   4.0026  /,
     *     ELEMNT(3),  ATWT(3)  / 'Li',   6.939   /,
     *     ELEMNT(4),  ATWT(4)  / 'Be',   9.0122  /,
     *     ELEMNT(5),  ATWT(5)  / 'B',   10.811   /,
     *     ELEMNT(6),  ATWT(6)  / 'C',   12.01115 /,
     *     ELEMNT(7),  ATWT(7)  / 'N',   14.0067  /,
     *     ELEMNT(8),  ATWT(8)  / 'O',   15.9994  /,
     *     ELEMNT(9),  ATWT(9)  / 'F',   18.9984  /,
     *     ELEMNT(10), ATWT(10) / 'Ne',  20.183   /
      DATA ELEMNT(11), ATWT(11) / 'Na',  22.9898  /,
     *     ELEMNT(12), ATWT(12) / 'Mg',  24.312   /,
     *     ELEMNT(13), ATWT(13) / 'Al',  26.9815  /,
     *     ELEMNT(14), ATWT(14) / 'Si',  28.086   /,
     *     ELEMNT(15), ATWT(15) / 'P',   30.9738  /,
     *     ELEMNT(16), ATWT(16) / 'S',   32.064   /,
     *     ELEMNT(17), ATWT(17) / 'Cl',  35.453   /,
     *     ELEMNT(18), ATWT(18) / 'Ar',  39.948   /,
     *     ELEMNT(19), ATWT(19) / 'K',   39.102   /,
     *     ELEMNT(20), ATWT(20) / 'Ca',  40.08    /
      END

      END
```

Example 2

A part of SFTRAN listing of DEMO produced with NUMBER=Y is as follows:

```
0008700 C.....MORE DETAILS:
0008800
0008900 30002 PROCEDURE (INITIALIZE FOR NEW FORMULA)
0009000 30002    MOLWT=0.0
0009100          LEVEL=0
0009200          NEXT=1
0009300          ERROR=.FALSE.
0009400 10008 END
0009500
0009600 30008 PROCEDURE (ASSEMBLE ELEMENT SYMBOL)
0009700 30008    DO (PUT FIRST BYTE INTO SYMBOL)
0009800 20037    DO (GET NEXT BYTE FROM FORMULA)
0009900 20038    IF (BYTE.GE.SMALLA .AND. BYTE.LE.SMALLZ) THEN
0010000             NEXT=NEXT+1
0010100 20039    ELSE
0010200 20039       BYTE=SPACE
0010300 20040    END
0010400 20040    DO (PUT SECOND BYTE INTO SYMBOL)
0010500 20041 END
0010600
0010700 30009 PROCEDURE (FIND MATCHING ELEMENT IN TABLE)
0010800 30009    DO FOR ATNO=1,NELEMS
0010900 20043       FOUND=SYMBOL.EQ.ELEMNT(ATNO)
0011000 10009    <--IF (FOUND) EXIT (ATNO)
0011100 20044    OTHERWISE
0011200 20044       ERROR=.TRUE.
0011300             WRITE (6,604) SYMBOL
0011400 20042    END
0011500 20045 END
0011600
0011700 30010 PROCEDURE (READ NUMBER OF ATOMS/RADICALS)
0011800 30010    N=0
0011900 20046    DO WITH
0012000 20046       DO (GET NEXT BYTE FROM FORMULA)
0012100 20048    WHILE (BYTE.GE.ZERO .AND. BYTE.LE.NINE)
0012200             N=10*N+(BYTE-ZERO)
0012300             NEXT=NEXT+1
0012400 10010    END
0012500 20047    N=MAX0(N,1)
0012600 10011 END
```

Example 3

A part of FORTRAN code produced from DEMO with NUMBER=Y is as follows:

```
C.....MORE DETAILS:                                                    8700
                                                                       8800
C     PROCEDURE (INITIALIZE FOR NEW FORMULA)                           8900
30002 MOLWT=0.0                                                        9000
      LEVEL=0                                                          9100
      NEXT=1                                                           9200
      ERROR=.FALSE.                                                    9300
10008 GO TO NPR002, (20003)                                            9400
                                                                       9500
C     PROCEDURE (ASSEMBLE ELEMENT SYMBOL)                              9600
30008 ASSIGN 20037 TO NPR011                                           9700
      GO TO 30011                                                      9701
20037 ASSIGN 20038 TO NPR007                                           9800
      GO TO 30007                                                      9801
20038 IF (.NOT.(BYTE.GE.SMALLA .AND. BYTE.LE.SMALLZ)) GO TO 20039      9900
      NEXT=NEXT+1                                                     10000
      GO TO 20040                                                     10100
20039 BYTE=SPACE                                                      10200
20040 ASSIGN 20041 TO NPR012                                          10400
      GO TO 30012                                                     10401
20041 GO TO NPR008, (20023)                                           10500
                                                                      10600
C     PROCEDURE (FIND MATCHING ELEMENT IN TABLE)                      10700
30009 ATNO=1                                                          10800
      N20042=NELEMS                                                   10801
      GO TO 20043                                                     10802
20042 ATNO=ATNO+1                                                     10803
      IF ((N20042-ATNO).LT.0) GO TO 20044                             10804
20043 FOUND=SYMBOL.EQ.ELEMNT(ATNO)                                    10900
10009 IF (FOUND) GO TO 20045                                          11000
      GO TO 20042                                                     11100
20044 ERROR=.TRUE.                                                    11200
      WRITE (6,604) SYMBOL                                            11300
20045 GO TO NPR009, (20024)                                           11500
                                                                      11600
C     PROCEDURE (READ NUMBER OF ATOMS/RADICALS)                       11700
30010 N=0                                                             11800
20046 ASSIGN 20048 TO NPR007                                          12000
      GO TO 30007                                                     12001
20048 IF (.NOT.(BYTE.GE.ZERO .AND. BYTE.LE.NINE)) GO TO 20047         12100
      N=10*N+(BYTE-ZERO)                                              12200
      NEXT=NEXT+1                                                     12300
10010 GO TO 20046                                                     12400
20047 N=MAX0(N,1)                                                     12500
10011 GO TO NPR010, (20027,20034)                                     12600
```

REFERENCES

1. SPERRY UNIVAC 1100 Series, FORTRAN (ASCII), Programmer Reference. UP-8244, Rev. 1, Sperry Rand Corp., 1976.

2. SPERRY UNIVAC 1100 Series, Conversational Time Sharing (CTS) System, Programmer Reference. UP-7940, Rev. 3, Sperry Rand Corp., 1976.

13 1 1U,G, 032778 S00903DS
DEPT OF THE AIR FORCE
AF WEAPONS LABORATORY
ATTN: TECHNICAL LIBRARY (SUL)
KIRTLAND AFB NM 87117